Software de Comunicaciones

4º Curso, segundo cuatrimestre

Profesores:
Juan Quemada, Joaquín Salvachúa
Dept. Ingeniería de Sistemas Telemáticos

La Asignatura

La Asignatura

- Objetivo:
  - Comprender los conceptos y las técnicas utilizadas en el diseño y realización de software de comunicaciones
- Contenido:
  - Servicios genéricos de comunicación
    - integrados en el S.O.
    - delimitados por el nivel de transporte
  - Aplicaciones distribuidas
    - accesibles al usuario
- La asignatura se basa en Internet
  - dada su amplia difusión

Lenguaje de Soporte

- Asignatura basada en **Java**
  - Lenguaje de descripción de algoritmos y programas
    - Es sencillo y fácil de aprender
    - Incluye conceptos actuales de programación
    - Es una norma de facto en Internet
  - Java incluye múltiples librerías
    - Sencillas para utilizar
    - Cubren los servicios de comunicación de Internet

Estructura de la asignatura

Se compone de tres partes diferenciadas:

- 1) Introducción al **lenguaje Java**:
  - Introducción y conceptos básicos (~1/6).
- 2) **Software de comunicaciones**:
  - Descripción de los conceptos y técnicas de diseño de software de comunicaciones (~1/2)
- 3) **Practicas** en el laboratorio:
  - Profundización en los conceptos desarrollados (~1/3)

Calendario (provisional)

0	**Introducción a la asignatura**	**1**	**Introducción a Java, Objetos**
2	**Herencia, Arr., Exc. e Interfaz**	**3**	**JFC, Canales y Streams**
4	**Internac., DNS y Mod. C-S**	**5**	**Sockets TCP**
6	**Concurrencia**	**7**	**Lab**
8	**Sockets UDP**	**9**	**Lab**
10	**Difusión y multicast**	**11**	**Lab**
12	**Modelado de Protocolos**	**13**	**Lab**
14	**El protocolo TFTP**	**15**	**Lab**
16	**Implementacion de TFTP**	**17**	**Lab**
18	**Serialización**	**19**	**Lab**
20	**RMI**	**21**	**Lab**
22	**Nivel de aplicacion**	**23**	**Lab**
24	**Applets y Servlets**	**25**	**Lab**
26	**Diseño de applicaciones**	**27**	**Lab**
28	**Repaso y dudas**	**29**	**Lab**

Laboratorio

- La asignatura incluye ~20 horas de laboratorio
- Las prácticas se proponen en las horas de teoría y se realizan en el laboratorio
  - Laboratorio: B 123
  - El acceso es libre: reserva previa (ver servidor web)
- Las practicas propuestas son independientes de una maquina y S.O.
  - se pueden realizar en cualquier plataforma que soporte JDK 1.1 o equivalente, como Windows 95 o Linux

Evaluación de la asignatura

- La asignatura se evaluará en un examen al final del cuatrimestre.
  - El examen incluirá problemas sobre las practicas de los temas
    - Socket, Remote Method Invocation y Aplicación
- Se deberá **adjuntar al examen** el **listado** de las **practicas incluidas en problemas**

Documentación

- La asignatura trata sobre un tema nuevo
  - no existe un único texto de referencia
    - Se ha seleccionado un soporte básico
- Java esta evolucionando
  - La asignatura se basa en Java 1.1
    - Ya esta disponible Java 1.2 (2.0), 1.3
  - Soporte documental hetereogeneo y dinámico
    - transparencias, libros, web, news, ...

Documentación basica: SWCM

- Transparencias
- Java Network Programming, Elliotte Rusty, Harold, O´Reilly 1997
- Servidor de “Software de Comunicaciones”
  - http://www.lab.dit.upm.es/~scom/
- Documentacion de JDK 1.1 (en HTML)
  - Describe todos los paquetes y JFCs de Java
- Una Introducción a Java
  - Programacion en Java, E. Walsh, Anaya Mult.
  - Java in the Nutshell (3d Ed.), D. Flanagan, O´Reilly, 99
    - para conocedores de C
    - De luxe Edition: incluye 5 libros en CDROM (para profundizar)

Documentación basica: SWCM

- Java Network Programming, Elliotte Rusty Harold, O´Reilly 1997 (Java 1.1)
  - Libro sobre programación de aplicaciones distribuidas para internet.
    - Cubre un alto porcentaje de la asignatura (60%)
    - Cubre los paquetes **java.net** y **java.rmi** (API comunicaciones)
    - Incluye: modelo cliente-servidor, DNS, sockets (TCP, Datagramas, Multicast), Servlets, RMI, acceso a web, URLs, manejadores de contenidos y de protocolos, CGIs

Programación de red - Java (II)

- Java Network Programming 2nd Ed., M. Hughes, M. Shoffner, D. Hammer, 1999 (1.2, 2.0).
  - Incl.: java.io, java.net, java.rmi, java.corba, java.servlets
  - Incluye: streams, serialización de objetos, modelo cliente-servidor, DNS, sockets (TCP, Datagramas), RMI, acceso a WWW, manejo de URLs, Corba, Servlets y aplicaciones diversas
  - La segunda edición del libro esta muy actualizada y cubre bien la asignatura.

Bibliografía complementaria

Información addicional: Java (I)

- La Agenda: Contiene información actualizada sobre internet (en castellano)
  - http://www.areas.net/agenda.htm
- Café au Lait: tutoriales, grupos de noticias y discusión, informaciones diversas
  - http://sunsite.unc.edu/javafaq/
- Servidores de SUN Microsystems sobre Java
  - http://www.javasoft.com/ (WWW Javasoft)
  - http://www.sun.com/edu/ (WWW SUN)
- Thinking in Java, Bruce Eckel, (Libro en PDF)
  - http://www.EckelObjects.com/javabook.html
  - Libro avanzado sobre Java, para profundización

Información adicional: Java (II)

- Gamelan: uno de los depositos de software preferidos en internet
  - http://www.gamelan.com/
- JARS: deposito de codigo, un jurado selecciona
  - http://www.jars.com/
- Java Language Newsgroup: grupo oficial de Java
  - news:comp.lang.java (muy extenso)
- Digital Expresso: resumen de Java Lang. Newsgr.
  - http://www.io.org/~mentor/DigitalExpresso.html
- Java FAQ Archives: preguntas mas frecuentes
  - http://www.net.com/java/faq

Otros libros: Java (castellano)

- Como Programar en Java (1.0), Deitel y Deitel, Prentice Hall 1998.
  - Libro de introducción a Java usado en Prog. (1er curso)
- Los secretos de Java a tu alcance (1.1), E. Rusty Harold, Anaya Multimedia, (5995 Pts)
  - Aspectos no documentados de Java (profundización)
- Java: Manual de Referencia, Naughton y Schildt, Osborne/Mc Graw Hill, 1997
  - Descripción del lenguaje por uno de sus diseñadores

Libros sobre Java (ingles)

- Java in the Nutshell (3rd Edition) : (Java 2), D. Flanagan, O´Reilly & Asociates Inc. ISBN: 1-56592-262-x.
  - Buena introducción a Java para programadores C
  - Version especial con **5 libros sobre Java en CDROM**
- Java: The complete Reference, Naughton & Schildt, Sun 1997
  - Guia de referencia escrita por uno de sus diseñadores

Bibliografía: Protocolos y Redes

- Computer Networks, A. S. Tanenbaum, Prentice Hall 1997.
  - Libro básico sobre protocolos y redes de ordenadores.
- Douglas Comer. Internetworking with TCP/IP (2nd ed.). Prentice Hall International, 1991 (L6c/Com).
  - Libro avanzado sobre los protocolos de internet.
- TCP/IP Illustrated, Volumes 1, 2 y 3, W. Richard Stevens, Addison Wesley 1994 (L6d/Ste).
  - Libro avanzado y completo sobre internet.
    - recomendado para profundizar

Introducción a las redes e Internet

Redes de Ordenadores

- Aparecen durante la década de los sesenta.
  - **Arpanet**: proyecto militar
    - Semilla de la cual surge **Internet**
  - UUCP (red de interconexión entre sistemas UNIX)
  - Comerciales:
    - Mainframes: SNA (IBM), DECNET (Digital), ...
    - PCs: Novel-Netware IPX
  - Modelo de referencia **MARISA** (OSI-ISO)
- Interconectan ordenadores
  - basadas en una nueva tecnología
    - conmutación de paquetes (almacenamiento y envío)

Aplicaciones distribuidas

- Servicios de red
  - Inicialmente emulan servicios locales del S.O.
    - Correo electrónico
    - Terminal virtual
    - Transferencia de ficheros
  - Al generalizase Internet aparecen servicios nuevos
    - Hipertexto y **WWW** (**World Wide Web**)
      - – permite conectar repositorios distribuidos de información
    - Aplicaciones para la colaboración: News, groupware, ...
    - Difusión de información: Webcast, PUSH, ..
    - Audio/Video conferencia: MBONE, ISABEL, H323, ...
    - .....

Historia de Internet

- Desarrollo inicial financiado por ARPA en los 60
  - Objetivo militar:
    - arquitectura de **red resistente a fallos** en los nodos
    - independiente de maquina
    - La mayoria de los desarrollos son de tipo “free software”
- Primer resultado: ARPANET
  - arranca con **4 nodos** en diciembre 1969
    - semilla donde se experimentan los protocolos de Internet
  - Acepta solo universidades con proyectos militares

Historia de Internet (cont.)

- NSF crea CSNET (finales de los 70)
  - Servicio a todos los grupos de investigación
  - Basada en acceso telefónico
- **Normalización de TCP/IP** (Enero 83)
- NSF crea NSFNET (1984)
  - conecta centros de supercomputación de NSF con ARPANET
- 1990: comienza crecimiento exponencial
  - en 1990 hay 3000 redes con 200.000 ordenadores.
- 1992: se conecta el ordenador 1 millón.
- 1995: Internet comienza su expansión social

¿Quien dirige Internet?

- ISOC: Internet Society
  - **Promueve el intercambio mundial de información**
- IAB: Internet Advisory Board
  - **Consejo de sabios que sanciona normas y propuestas**
- IETF: Internet Engineering Task Force
  - **Conjunto de comités técnicos de asistencia voluntaria**
    - Grupos de trabajo para cada RFC (Request for Comment)
      - – Trabajan con una enorme libertad
        - » Las normas no se imponen
        - » Los usuarios acaban seleccionando las propuestas
      - – Muy relacionado con el mundo del “free software”

Historia del WWW

- 1989: Tim Berners-Lee (CERN)
  - propone un sistema hipertexto para
    - coordinar experimentos de física de altas energias con Internet
- 1991: Hypertext´91, San Antonio - Texas
  - demostración publica de un visor para texto
- 1993: primer visor gráfico (Mosaic de NCSA).
- 1994: CERN y M.I.T. crean el WWW Consortium
- 1995: aparece Netscape
  - Fundada por Marc Andreessen
    - desarrollador de Mosaic en NCSA
- 1996: Microsoft se vuelca en Internet
  - Microsoft desarrolla “Explorer” (a partir de Mosaic)

Historia de Java

- 1991: Green Project (Naughton, Gosling y Sheridan)
  - Desarrollar un lenguaje (OAK) para carga y mantenimiento de dispositivos remotos (SUN)
    - Para entornos domésticos de vídeo bajo demanda
    - Uniendo ideas de C, C++ y Smalltalk (OO)
  - El proyecto fracasa
    - SUN no es seleccionado en la experiencia de Orlando
- 1994: Cambio de rumbo
  - Adaptación de la tecnología a Internet: **Java**
    - Programación distribuida en un entorno WWW
    - Programas cargables a traves de la red: **Applets**

Conclusión

- Internet ha modificado la forma de hacer tecnología
  - La red permite una comunicación mucho mas rápida y eficaz en un entorno global
    - Web:
      - cambia el paradigma de publicación de información
    - Groupware:
      - cambia los métodos de trabajo
  - Todos los procesos se aceleran y flexibilizan
    - Internet 30 años, WWW 8 años, Java 4 años, ...
- Objetivo SWCM:
  - Introducir el desarrollo de aplicaciones y servicios en este entorno cambiante y distribuido

Introducción a JAVA

Lenguaje orientado a la red

- Pensado para ser usado en la red
  - incluye seguridad
  - permite cargar objetos a través de la red
- Independiente de maquina: multiplataforma
  - Permite portar Java a cualquier procesador y/o S.O. con poco esfuerzo
    - Solo hay que implementar un interprete
- Integrado en WWW
  - Visores implementan la Maquina Virtual Java (JVM)
    - Netscape, Microsoft Explorer, Hotjava, ...
  - Cargan objetos del servidor para ejecutar en cliente

Caracteristicas de Java

- Lenguaje independiente de maquina
  - Definición basada en maquina virtual interpretada
    - Fuentes de disponibles en C
    - Lenguaje de la maquina virtual: Bytecodes
      - Equivalente al lenguaje maquina de un procesador
    - Formato de fichero maquina: Class file format
- Orientado a objetos: fuertemente tipado
- Gestión automática de memoria
  - No hay punteros, solo referencias a objetos
- Multihebra (procesos ligeros)

Programas en Java

- Clase: elemento central de un programa Java
- Una clase se encapsula en un fichero
  - El fichero tener el mismo nombre de la clase.
- Una clase puede invocarse como una aplicación si tiene un método
  - public static void main(String args**[]**)
    - Los parametros se pasan en: **args[]**
- La ejecución de la clase finaliza al acabar main
  - No devuelve parametro de terminación

Ejemplo: aplicacion HelloWorld

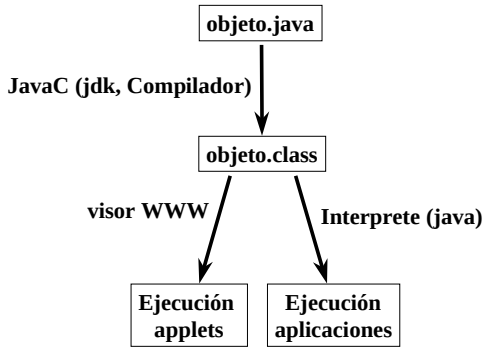
- Modificador “public”
  - Significa accesible en cualquier ámbito
- Para imprimir
  - System.out.println(.....)

```
public class HelloWorld {

   public static void main (String args[]){

      System.out.println("Hello World!");
   }
}
```

JDK

- Compilar y ejecutar el programa HelloWorld (JDK)
  - JDK se ejecuta desde Linux/MSDOS
    - JDK da algún problema en Linux
- >javac HelloWorld.java // compila la clase
- >java HelloWorld // ejecuta la clase
- **Ademas en MSDOS (AUTOEXEC.BAT)**
  - **PATH C:...\jdk1.1.3\bin**
  - **doskey**

Niveles de lenguaje



Ejercicio Java 1

- Los “Strings” se concatenan en Java con el operador “+”.
- Los argumentos de la invocación de la clase se pueden leer como si fuesen campos de el array “args**[]**”. El primer argumento es el numero “0”.
- Se pide modificar el programa Hello World para que en vez de “Hello World!” imprima “Hola este es el primer argumento: <primer argumento>!”.

Tipos y Objetos

- En Java hay tres clases de tipos
  - Tipos primitivos: booleanos, enteros, ...
    - tipos predefinidos en el lenguaje con sintaxis especial
    - se manejan siempre por valor
  - Clases y Objetos
    - definibles por el usuario
    - Se manejan siempre por referencia
  - Arrays (matrices)
    - Similares a los objetos
      - sintaxis especial

Tipos predefinidos, operadores y sentencias

Tipos predefinidos I

Tipo	Contenido	Valor por def.	Bits	Valores min. y max.
boolean	true o false	false	1 bit	
char	caracter Unicode	\u0000	16 bits	\u0000 \uFFFF
byte	Complemento a 2	0	8 bits	-128 127
short	Complemento a 2	0	16 bits	-32768 32767

Tipos predefinidos II

Tipo	Contenido	Valor por def.	Bits	Valores min. y max.
int	Complem. a 2	0	32 bits	-2147483648 2147483647
long	Complem. a 2	0	64 bits	-9223372036854775808 9223372036854775807
float	IEEE 754	0.0	32 bits	±1.40239846 E-45 ±3.40282347 E+38
double	IEEE 754	0.0	64 bits	±4.94065645841246544 E-324 ±1.7976931348623157 0 E+308

Tipos primitivos: ejemplos

```
boolean   on_the_floor = true;
              // crea la variable on_the_floor e inicializa a true
boolean under;  // crea under, Java asigna false por defecto

on_the_floor = false;      // asigna false

under = true;              // asigna true

int count = 7;          // crea la variable count, inicializa a 7

count = count + 1;   // incrementa count
count++;             // incrementa count (postincremento)
++count;             // incrementa count (preincremento)
count += 1;          // incrementa count
```

Tipos primitivos: char

```
char first = 'A';       // crea first y asigna caracter: A

char c = '"';           // crea c y asigna caracter: "
c = '\u0022';           // asigna caracter: "
c = '\042';             // asigna caracter: "
c = '\"';               // asigna caracter: "
      // 22 (hexadecimal) = 42 (octal) = código de caracter "

       //  \b    espacio atrás
       //  \t    tabulador horizontal
       //  \n    linea nueva
       //  \r    retorno de carro
       //  \f    paso de pagina
       //  \"    comillas
       //  \'    comilla
       //  \\    barra hacia atras
```

Sentencia: while

- El bucle while analiza la condición antes de ejecutar el cuerpo del bucle
  - el programa imprime los argumentos por consola

```
class PrintArgs1 {

  public static void main (String args[]){
    int i = 0;
    while (i < args.length) {
      System.out.println ("argumento " + i + ": " + args[i++]);
    }
  }
}
```


Sentencia: for

- Inicializa (y define) variables antes de comenzar
  - Permite varias sentencias separadas por comas
- Verifica la condición antes de ejecutar el bucle
- Actualiza variables al final de cada ejecución
  - Permite varias sentencias separadas por comas

```
class PrintArgs4 {

  public static void main (String args[]){
    for (int i = 0; i <args.length; i++) {
      System.out.println ("argumento " + i + ": " + args[i]);
    }
  }
}
```

Tipos predefinidos, operadores y sentencias

(Transparencias complementarias)

Sentencia: do-while

- El bucle do-while se ejecuta al menos una vez
  - este programa activa una excepción si args.length = 0

```
class PrintArgs2 {

  public static void main (String args[]){
    int i = 0;
    do {
      System.out.println (“argumento “ + i + “: “ + args[i++]);
    } while  (i < args.length);
  }
}
```

Sentencia: break

- break aborta la ejecución: si args.length = 0
  - Utilizable con: while, for, do-while y switch

```
class PrintArgs3 {

  public static void main (String args[]){
    int i = 0;
    do {
      if (args.length == 0) { break;};     // aborta la ejecucion
      System.out.println (“argumento “ + i + “: “ + args[i++]);
    } while  (i < args.length);
  }
}
```


Sentencia: continue

- “continue” finaliza la ejecución de la iteración
  - Utilizable con: while, for y do-while
- Printhola imprime los argumentos igual a “hola”

```
class PrintHola1 {

  public static void main (String args[]){

    for (int i = 0; i <args.length; i++) {
      if (!args[i].equals(“hola”)) then { continue; }
      System.out.println (“argumento “ + i + “: “ + args[i]);
    }
  }
}
```

Sentencia: try/catch

- Excepción: interrumpe la ejecución secuencial
  - puede ser capturada dentro de una sentencia try/catch
- Ejemplo: Printhola con excepciones

```
class PrintHola2 {
  public static void main (String args[]){
    for (int i = 0; i <args.length; i++) {
      try {
        if (!args[i].equals(“hola”))
                    { throw new java.lang.Exception(); }
        System.out.println (“argumento “ + i + “: “ + args[i]);
      } catch (Exception e) {  System.out.println (e); }
    }
  }
}
```

Excepciones

- La activación interrumpe la ejecución secuencial
  - se propagan hasta que son capturadas
    - interrumpe toda sentencia que no la capture
  - la ejecución continua con el código del primer catch que acepte la excepción activada
- Pueden ser activadas por un programa
  - con la sentencia **throw**
- La maquina virtual Java las puede activar también
- Un método debe declarar sus excepciones
  - int div (int op1, int op2) **throws someEx** (.....)

Sentencia: switch

```
int llave;
......
switch (llave) {
   case 3:
      .......
      break;
   case 2:
      .......
      break;
      .......
   case <valor>:
      .......
      break;
   default:
      .......
      break;
}
 ......
```

- llave puede ser de tipo
  - byte, char, short, int, long
- Ejecuta a partir del código del caso que cumple
  - llave == <valor>
- Hay que insertar break para no ejecutar los siguientes casos

Etiquetas

- etiquetas permiten asociar bucles y sentencias anidadas a break y continue

```
uno:  if (cont[x,y] == true) {
buc:    for (int i = 0; i <max; i++) {
          for (int j = 0; j <max; j++) {
            if (i == 7) then { continue buc; }
                    // aborta iteracion de bucle (pasa a: i = 8)
            if (cont[i,j] == false) then { break uno; }
                                    // aborta ejecucion "if"
            System.out.println ("argumento " + i);
          }
        }
        System.out.println ("Final de bucle");
      }
```

Operandos I

Prior/ Asoc	ope-rador	operandos	descripcion
1, R	++, --	aritmetico	pre/post incr/decremento
1, R	+, -	aritmetico	mas, menos (unitario)
1, R	~	entero	complemento bit a bit
1, R	!	boolean	complemento logico
1, R	(tipo)	any	cast - conversion de tipos
2, L	*, /, %	aritm., aritm.	multiplicacion, division, resto
3, L	+, -	aritm., aritm.	suma, resta
3, L	+	string, string	concatenacion de strings
4, L	<<	entero	desplazamiento a la izquierda
4, L	>>	entero	despl. derecha, insercion signo
4, L	>>>	entero	despl. derecha, insercion cero

Operandos II

Prior/ Asoc	operador	operandos	descripcion
5, L	<, <=	aritmetico, arit.	menor, menor o igual
5, L	>, >=	aritmetico, arit.	mayor, mayor o igual
5, L	instanceof	Object, class	prueba de tipo
6, L	==, !=	primitivo, primiti.	igual, diferente (valor)
6, L	==, !=	Object, Object	igual, diferente (puntero)
7, L	&	entero,entero	and (bit a bit)
7, L	&	boolean, bool.	and (logico)
8, L	^	entero, entero	xor (bit a bit)
8, L	^	boolean, bool.	xor (logico)
9, L	|	entero, entero	or (bit a bit)
9, L	|	boolean, bool.	or (logico)

Operandos III

Prior/ Asoc	operador	operandos	descripcion
10, L	&&	boolean, boolean	and condicional (evaluacion parcial hasta resolver expr.)
11, R	||	boolean, boolean	or condicional (evaluacion parcial hasta resolver expr.)
12, R	?:	boolean, any, any	if-then-else funcional

13, R	=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	any	asignacion con operacion
			x += 3;
			igual a
			x = x + 3;

Operaciones: ejemplos

```
static final int i = 88;     // define una constante entera
static final byte b = 3;   // define una constante de tipo byte

byte bb = (byte) i; // la función de cast adapta tipos (int a byte)

int val = b; // de menor a mayor no hace falta hacer cast

byte b = (3 < ++val)?3:2; // asigna 3, expresión = true

byte b = (byte) (value >> 8);  // extrae segundo byte de value

int i = (0x000000600 & intvalue) >>> 9;  // extrae un campo de
          // dos bits (bits 9 y 10) con una mascara hexadecimal
          // y calcula su valor numerico
```

Cadenas de caracteres (String)

- String: instancia de **java.lang.String**
  - El manejo de Strings con constructores y metodos de la clase es muy complejo
  - Sintaxis especial para manejo de String
    - Operador de concatenación: +
    - Constantes tipo String: “xxxxxxx”
  - Ejemplo: String s = “hola “ + var + toString(value);
- Las operaciones con strings se realizan con java.lang.StringBuffer
  - Un String no puede modificarse, solo copiarse
  - Las operaciones se realizan sobre StringBuffers

Ejercicio Java 2

- Realizar un programa que reciba varios argumentos
  - cuando el argumento sea 11, 12 o 13 imprima
    - "numero once", "numero doce" o "numero trece"
  - cualquier otro argumento debe imprimirlo sin modificación
- Utilizar la sentencia switch para realizarlo

Clases y objetos

Clases y objetos

- El mecanismo de definición de clases permite crear nuevos tipos de usuario
  - substituye un **struct** de C, o un **record** de Pascal
  - añade **control de visibilidad** y **herencia**
- Clase:
  - Conjunto de datos encapsulados con los métodos que permiten manipularlos de forma coherente
  - Una clase se compone de:
    - campos, constructores y métodos
- Objeto:
  - instancia de una clase
    - se crea y destruye dinámicamente

Ejemplo: definición de clase

- "public" significa visible (utilizable) en el exterior
- "private" significa privado de la clase
  - no es visible (utilizable) en el exterior

```
public class website {

   private String name;                          // campos
   private String url;
   private String descr;

   public website(String n, String u, String d)    // constructor
              {  name = n; url = u; descr = d; }

   public void print()                           // método
      { System.out.println(name + " at " + url + descr); }
}
```

Ejemplo: uso de una clase

- Un objeto se crea con un constructor
  - el objeto se destruye cuando deja de utilizarse
    - Es destruido automáticamente por la maquina virtual Java

```
class websiteTest {

  public static void main(String args[]){

     website w = new website(“Juan Quemada”,
         “http://www.dit.upm.es/quemada”,
         “pagina personal de J. Quemada”);       // creación

     w.print();                                  // uso
     System.out.println (“Name is: “ + w.name);
  }
}
```

Visibilidad

- Visibilidad: accesibilidad de un objeto o de sus componentes por otros objetos
  - En la definición de un objeto se puede controlar la visibilidad de sus componentes
    - con los atributos: public, private, ...

public - visible en cualquier lugar (máxima visibilidad)

private - visible solo dentro de la clase

.....

Creación de objetos

- Un objeto se maneja siempre por referencia
  - el objeto se puede construir en dos pasos
    - paso 1: creación de la variable de referencia
    - paso 2: creación del objeto asociado a la variable de referencia
  - ambos pasos se pueden unificar (ejemplo anterior)

```
website w;       // creacion de la variable de referencia w
                 // w se inicializa a “null”
                 // “null” representa un objeto no creado

w = new website(“Juan Quemada”,
    “http://www.dit.upm.es/quemada”,
    “pagina personal de J. Quemada”);
       // creacion del objeto asociado a la variable w
```

Ejercicio Java 3

- Extender la clase “website” con un modificador de cada campo (name, url, descr) y crear un programa de prueba que
  - 1) cree un objeto URL
  - 2) imprima su contenido
  - 3) modifique un campo
  - 4) imprima el contenido modificado

Sobrecarga de operadores

- Métodos y constructores se pueden sobrecargar
  - La sobrecarga (overloading) se resuelve en función de los tipos de los argumentos
    - Error: nombres iguales con la misma tupla de tipos de argumentos

```
public class websiteS {
  String name;  String url;  String descr;          // campos

  public websiteS(String n, String u, String d) // constructor 1
    { name = n; url = u; descr = d; }
  public websiteS(String n, String u)               // constructor 2
    { name = n; url = u;  }

  public void print()                                  // metodo 2
    { System.out.println(name + " at " + url + descr);  }
  public void print(String msg)                        // metodo 2
    { System.out.println(msg + name + " at " + url + descr);  }
}
```

Variables y metodos estaticos

- Métodos y variables estáticas pertenecen a la clase
  - Una variable estática es la misma en todos las instancias
  - Un método estático solo puede acceder a vars estáticas
    - Se invocan sobre la clase, no sobre las instancias

```
public class Swebsite {
  String name;  String url;  String descr;
  static int counter = 0;          //cuenta los objetos creados

  public Swebsite(String n, String u, String d)
    { name = n; url = u; descr = d; counter++; }

  public void print()
    { System.out.println(name + " at " + url + descr);  }

  static public int set(int i)       // metodo estatico
    { return counter += i;  }
}
```


Uso de elementos estaticos

- la variable counter es la misma en todos los objetos
  - Swebsite.set(0) devuelve primero un 1 y luego un 2

```
public class SwebsiteTest {
  public static void main(String args[]) {

    Swebsite s1 = new Swebsite ("Juan Quemada",
      "http://jq.dit.upm.es", "pagina personal");
    System.out.println("no de obj.: " + Swebsite.set(0));

    Swebsite s2 = new Swebsite ("Joaquin Salvachua",
      "http://jsr.dit.upm.es", "pagina personal");
    System.out.println("no de obj.: " + Swebsite.set(0));
  }
}
```

Clases y objetos

(Transparencias complementarias)

Elementos de una clase

- Cabecera: define su nombre y su tipo
- Variables o campos: contienen el estado
- Constructores: crean objetos de la clase
  - los objetos se denominan “instancias” de la clase
  - tienen el mismo nombre que la clase
  - pueden sobrecargarse.
- Metodos: operaciones realizables a un objeto
  - pueden sobrecargarse.

Constructor por defecto

- Constructor por defecto
  - Solo existe si la clase no posee constructores explícitos
  - No tiene parámetros: **<classname>()**
    - Campos del objeto creado se inicializan con valores por defecto
    - Es publico

```
website w = new website() ;
   // crea un objeto de tipo website
   // todos los campos (name, url, descr) se inicializan a null
```


Tapado de constructores

- website0 no posee constructores utilizables
  - El usuario no puede crear objetos de tipo website0
  - Pregunta: ¿Como se puede crear un objeto website0?

```
public class website0 {
  private String name;                          // campos
  private String url;
  private String descr;

  private website0(){}                  // constructor no utilizable

  public void print(){                          // metodo
    System.out.println(name + “ at “ + url + descr);
  }
}
```

Destrucción de objetos

- La maquina virtual Java destruye los objetos cuando no se utilizan mas
  - libera la memoria y el resto de recursos que se han asignado al crear el objeto
  - Hay otros recursos que se asignan en el codigo
    - descriptores de ficheros, sockets, .. no se liberarian
  - existe un metodo finalize
    - se invoca antes de destruir el objeto y permite realizar limpieza

```
public class ..... {
  FileInputStream fd;
  .....
  protected void finalize() throws IOException {
     { if (fd != null) fd.close();  }
}
```

Inicializadores estaticos

- Los campos estaticos no pueden ser inicializados en los constructores
  - Inicializador estatico: inicializa las variables estaticas
    - Se ejecuta al cargar la clase
  - Tambien hay inicializadores no estaticos
    - su utilidad es muy limitada

```
public class SomeClass {
  public static void main() {
     static int line[] = new int [20];

     static {
        for (int i=0; i < line.length; i++, line[i] = a*i +b) {}
     }

     .... SomeStaticMethods ....
  }
}
```

Estructuras Dinámicas

Estructuras dinámicas

- Los objetos se manejan a través de referencias
  - Una referencia es mas segura que un puntero
  - No existe forma de introducir valores inconsistentes
  - Una variable puede contener:
    - 1) null (antes de crear el objeto)
    - 2) una referencia a un objeto ya creado
      - Nunca referencia una zona de memoria inconsistente
  - La maquina virtual garantiza que no se pierde memoria
    - cuando un objeto deja de utilizarse su memoria se libera
- En Java se pueden crear estructuras dinámicas utilizando referencias como punteros

Estructuras dinamicas: Pila

- Implementación como una lista encadenada
  - Una pila se crea con el constructor **stack()**
  - La pila se maneja con **push(i)** y **pop()**

```
public class stack {
  private stack first;
  private int i;

  public stack () {}
  private stack (stack s, int j) { first = s; i = j; }

  public void push (int i)  { first = new stack (first, i); }
  public int pop () {
    stack s = first;   first = first.first;  return s.i;
  }
}
```

Prueba de la pila

- Programa de prueba
  - introduce tres números (1,2,3)
  - Extrae 4 elementos. El cuarto activa una excepción.

```
public class stackTest {

   public static void main(String arg[]) {
      stack s = new stack();
      s.push(1);
      s.push(2);
      s.push(3);
      System.out.println("primero: " + s.pop());
      System.out.println("segundo: " + s.pop());
      System.out.println("tercero: " + s.pop());
      System.out.println("cuarto: " + s.pop());
   }
}
```

Ejercicio Java 4

- Definir una clase que permita crear arboles binarios de forma recursiva a través de sus constructores (según esqueleto adjunto)
- Definir un programa de prueba que construya el árbol ((*,1,(*,4,*)),2, *) y lo imprima
  - * significa null

```
public class tree {
   tree left, right;
   int val;

  public tree (tree l, int v, tree r) { ..... }

  public print (tree t) { ..... }
}
```


Estructuras Dinámicas

(Transparencias complementarias)

Estructuras dinamicas: Cola

- Implementación: doble lista circular
  - no tiene punteros vacíos

```
public class queue {
   queue next, prev;
   int val;

   queue () { next = this; prev = this; }
   private queue(queue n,queue p,int v){ next=n;prev=p;val=v;}

   public void put (int i) {
      prev.next = new queue(prev.next, prev, i);
      prev = prev.next;
   }
   public int get () {

      queue q = next;
      next = next.next;
      next.prev = this;
      return q.val;
   }
}
```

Prueba de la cola

- Programa de prueba
  - introduce tres números (1,2,3)
  - Extrae 4 elementos. El cuarto devuelve 0 (valor pred.).

```
public class queueTest {

  public static void main(String arg[]) {
     queue s = new queue();
     s.put(1);
     s.put(2);
     s.put(3);
     System.out.println("primero: " + s.get());
     System.out.println("segundo: " + s.get());
     System.out.println("tercero: " + s.get());
     System.out.println("cuarto: " + s.get());
  }
}
```

Herencia

Herencia

- La herencia permite crear nuevas clases a partir de clases existentes
  - diseño incremental
- La herencia permite enriquecer clases
  - Añadir nuevas variables y metodos
- La herencia permite modificar clases
  - cambiar el significado de variables y/o metodos
- Terminologia y notación: Class B extends A { **....** }
  - B es un subtipo de A
  - B extiende A
  - A es la superclase de B

Ejemplo: website otra vez

```
public class website {

  private String name;                          // campos
  private String url;
  private String descr;

  public website(String n, String u, String d){    // constructor
     name = n; url = u; descr = d;
  }

  public void print(){                          // metodo
    System.out.println(name + " at " + url + descr);
  }
}
```

Herencia: enriquecimiento

- websiteE añade: variable **email**, metodo **printEmail**
  - Campos y metodos de website son visibles en websiteE
- Invocación constructor superclase en 1a sentencia
  - **super:** referencia al constructor de la superclase
  - **this:** referencia al objeto en construcción

```
public class websiteE extends website {
  public String email;                          // nuevo campo

  public websiteE(String n, String u, String d, String email)
    {super(n, u, d); this.email = email;}
                                                // nuevo constructor
  public void printEmail() {
    System.out.println
                (“To contact “ + name + “ send msg to “ + email);
  }                                             // nuevo metodo
}
```

Herencia: modificación

- websiteM modifica website
  - Añade tambien la variable: email
  - Pero redefine el significado de: **print()**

```
public class websiteM extends website {
  public String email;

  public websiteM(String n, String u, String d, String email)
    {super(n, u, d); this.email = email;}

  public void print() {
    super.print();
    system.out.println
                (“To contact “ + name + “ send msg to “ + email);
  }                                             // nuevo metodo
}
```


Programa de prueba

```
public class websiteEMTest {

 // invocacion: testWebsite <name> <url> <descr> <email>
 // crea tres objetos de clases websiteX e invoca sus
 // metodos print y printEmail en su caso.

   public static void main(String a[]){
      websiteE  wE  = new websiteE(a[0], a[1], a[2], a[3]);
      websiteM wM = new websiteM(a[0], a[1], a[2], a[3]);
     wE.print();            // invoca print de websiteE
     wE.printEmail();  // invoca printEmail de websiteE
     wM.print();            // invoca print de websiteM
   }
}
```

Uso y herencia de objetos

- Clases y objetos se pueden usar por otro objeto
  - si están en su ámbito de visibilidad
    - Usar una clase significa crear objetos o variables
    - Usar un objeto significa invocar métodos o asignar campos
- Una clase se puede heredar
  - si está en el ámbito de visibilidad

- ¿website usa o hereda alguna clase?
- ¿websiteE/M usan o heredan alguna clase?
- ¿websiteEMTest usa o hereda alguna clase?

Ejercicio Java 5

- Definir una clase que extienda websiteE añadiendo nuevos campos para los numeros de telefono, de fax y de movil.
  - Añadir un nuevo método “printTf()” que imprima
    - “Los números de contacto de “name” son:
      tf = <tf>, fax = <fax>, movil = <movil>

Jerarquia de clases

- La herencia en java es de tipo simple (o lineal)
  - Solo se hereda una clase
    - Una clase solo tiene una superclase
    - Una clase puede tener varias subclases
- Cualquier clase deriva de “java.lang.Object”
  - “java.lang.Object” esta predefinida en JFC
    - JFC (Java Foundation Clases)
  - “java.lang.Object” contiene metodos y variables que debe poseer cualquier clase
    - equals(), clone(), ...
  - Las clases forman una jerarquia cuyo vertice es “java.lang.Object”

java.lang.Object

```
public class Object {
   public boolean equals(Object obj);
   public final native Class getClass();
   public native int hashCode();
   public final native void notify();
   public final native void notifyAll();
   public String toString();
   public final native void wait(long timeout)
                              throws InterruptedException;
   public final void wait(long timeout, int nanos)
                              throws InterruptedException;
   public final void wait() throws InterruptedException;
   protected native Object clone()
                              throws CloneNotSupportedException;
   protected void finalize() throws Throwable;
}
```

Modificador “final”

- Clases: final de la herencia
  - no puede haber subclases
- Métodos: no modificables
  - una subclase no puede redefinir el método
- Variables: las transforma en constantes
  - su contenido no puede variar

```
final class websiteM extends website {
  public String email;

  public websiteM(String n, String u, String d, String email)
    {super(n, u, d); this.email = email;}

  final public void print() {
      system.out.println (“To .....+ email);
  }                          // metodo no modificable por herencia
}     // una clase derivada solo puede tener este metodo print()
```

Igualdad y clonación de objetos

- La variable de un objeto solo contiene una referencia al objeto
- Se pueden realizar copias y/o comparaciones
  - del objeto o de su referencia
- Igualdad
  - La comparación de objetos se realiza con: "equals()"
    - "equals" es un metodo heredado de java.lang.object
  - La comparación de variables compara las referencias
- Copia
  - La copia de un objeto se realiza con: "clone()"
    - "clone" es un metodo heredado de java.lang.object
    - El objeto debe implementar el interfaz java.lang.cloneable
  - La copia de variables copia las referencias

Ejemplos: Igualdad y clonación

```
website w1, w2, w3, w4;
w1 = new website("jq", "http://jq.upm.es/", "pag. personal");
w2 = new website("jq", "http://jq.upm.es/", "pag. personal");
w3 = w2;                  // copia la referencia de w2 en w4
w4 = w2.clone();          // crea una copia nueva de w2 en w4

boolean r1, r2, r3, r4, r5, r6;
r1 = (w1 == w2);          // asigna false
r2 = w1.equals(w2);       // asigna true

r3 = (w2 == w3);          // asigna true
r4 = w2.equals(w3);       // asigna true

r5 = (w2 == w4);          // asigna false
r6 = w2.equals(w4);       // asigna true
```


Herencia

(Transparencias complementarias)

Herencia: enriquecimiento bis

- Si la subclase no incluye el constructor de superclase como primera sentencia de algún constructor
  - El constructor invoca automáticamente el constructor por defecto de la superclase antes de la primera sentencia

```
public class websiteEb extends websiteD {
  public String email;                          // nuevo campo

  public websiteEb(String n, String u, String d, String e)
    {name = n; url = u; descr = d; email = e;}
                                                  // nuevo constructor
  public void printEmail() {
    System.out.println("To contact "+url+" send msg to "+email);
  }                                               // nuevo metodo
}
```

websiteD

- websiteD debe tener un constructor sin argumentos
  - websiteD() es invocado por el constructor de websiteEb
    - Un objeto de una subclase debe construir siempre el objeto de su superclase antes de crearse

```
public class websiteD {

  String name; String url; String descr;          // campos

  public websiteD () {}
  public websiteD (String n, String u, String d)
        {name = n; url = u; descr = d;}          // constructores

  public void print(){                            // metodo
    System.out.println(name + " at " + url + descr);
  }
}
```

Referencias this y super

- **this:** tiene dos usos basicos
  - referencia al objeto
    - Un metodo no estatico siempre incluye una referencia **this** ademas de sus parametros
    - Esta referencia se utiliza para acceder a campos del objeto o invocar metodos sobre el
    - cuando es redundante **this** se puede omitir
  - referencia al nombre del constructor de la clase
    - debe estar siempre en la primera sentencia de otro constructor
- **super:** tiene dos usos basicos
  - referencia a un metodo tapado de la superclase
  - referencia al nombre del constructor de la superclase
    - debe estar siempre en la primera sentencia de otro constructor

Herencia de constructores

- Al crear un objeto derivado de otro es necesario construir el objeto de forma incremental
  - Invocando a los constructores de las superclases
- Java garantiza la invocación de todos los constructores cuando se crea un objeto
  - Constructor implícito
    - llama a constructores por defecto (empieza por supertipo)
  - Constructor explícito
    - debe invocar a constructor explícito del supertipo en la primera línea
    - si no se invoca automaticamente el constructor implícito antes de ejecutar el constructor
      - excepción: antes de invocar el constructor del supertipo se pueden invocar otros constructores del mismo tipo

Matrices (Arrays)

Matrices (Arrays)

- Definen estructuras regulares de información
  - Ejemplos:
    - byte PDU[] = new byte[4];
    - byte PDU[] = {2, 4, 8, 9}; // utilizable solo en inicialización
- Su manejo es similar a los objetos
  - Se manejan por referencia
  - Se crean dinámicamente (con new)
  - Se liberan de memoria cuando dejan de usarse
- Arrays derivan de Object
  - pueden ser asignados a variables de tipo Object

Ejemplos: matrices

```
// definición

public byte[] PDU(int type, int len, byte[] data);

String buffer[];
int[] a, b[];       // a es unidimensional, b es bidimen.

// uso

byte[] datos = {0,0,0,0};  // esta sintaxis solo es
          // valida en la creación, en asignaciones y
          // otros usos se debe utilizar arrays dinámicos
byte[] buff = PDU(8, 10, datos);

int i = buff.length

Object o = {0,0,0,0};
```


Matrices (Arrays)

(Transparencias complementarias)

Matrices (II)

- El tamaño de un array no es parte del tipo
  - una variable de tipo array puede contener arrays de diversas longitudes
- Un array puede ser multidimensional
  - arrays multidimensionales se implementan como arrays de arrays
    - byte PDU[][][] = new byte[4][4][4];
- desde Java 1.1 existen arrays anónimos
  - no tienen nombre

Ejemplos: matrices (II)

```
// definición e inicialización de array multidimensional
// cada array unidimensional es de tamaño diferente
// es posible porque el tamaño no pertenece al tipo

byte buffer[][] = { { 1 },
                    { 0, 2 },
                    { 2, 3, 1 },
                    { 0, 4, 1, 1 },
                    { 0, 6, 1, 1, 8 },
                    { 0 }
                  };

// array anónimo, no tiene nombre

byte[] buff = PDU(8, 10, new byte[] {0,0,0,0,0,0,0});

s = new int[] {0,1,3,5,7,11,13};
```

Excepciones

Excepciones

- Mecanismo de interrupción de la ejecución secuencial
  - Se han considerado como una variante mas estructurada de la sentencia go-to
- Una interrupción es una señal
  - Al producirse interrumpe la ejecución secuencial
    - En un entorno fuertemente tipado las interrupciones solo se pueden producir en el ámbito en que han sido declaradas
  - La ejecución continua cuando la interrupción es capturada

Sentencia: try/catch

- Excepción: interrumpe la ejecución secuencial
  - puede ser capturada dentro de una sentencia try/catch
- Ejemplo: Printhola con excepciones

```
class PrintHola2 {
  public static void main (String args[]){
    for (int i = 0; i <args.length; i++) {
      try {
        if (!args[i].equals("hola"))
                    { throw new java.lang.Exception(); }
        System.out.println ("argumento " + i + ": " + args[i]);
      } catch (Exception e) {  System.out.println (e); }
    }
  }
}
```

Excepciones: operativa

- La activación interrumpe la ejecución secuencial
  - se propagan hasta que son capturadas
    - interrumpe toda sentencia que no la capture
  - la ejecución continua con el código del primer catch que acepte la excepción activada
- Pueden ser activadas por un programa
  - con la sentencia **throw**
- La maquina virtual Java las puede activar también
- Un metodo debe declarar sus excepciones
  - int div (int op1, int op2) **throws someEx** (.....)

Excepciones: jerarquia

- Las excepciones son objetos
  - el usuario puede definirse nuevas excepciones
- Todas derivan de la clase: java.lang.Trowable
  - java.lang.Trowable tiene dos subtipos
    - Error y Exception
- java.lang.Exception
  - supertipo de donde derivan las excepciones
    - suelen indicar condiciones recuperables
- java.lang.Error
  - supertipo de donde derivan todos los errores
    - suelen indicar condiciones no recuperables

Declaración de excepciones

- Las excepciones se declaran como cualquier objeto
  - Suelen tener un constructor con argumento String para poder enviar un msj de depuración al activarla

```
class Exception1 extends Exception {
   public Exception1 () {  super(); }
   public Exception1 (String s) {  super(s); }
}

class Exception2 extends Exception {
   public Exception2 () {  super(); }
   public Exception2 (String s) {  super(s); }
}
```

Excepciones

(Transparencias complementarias)

Sentencia: try/catch/finally

- La sentencia completa de captura de excepciones posee tres partes:
  - **try/catch/finally**
- try
  - contiene el codigo principal
    - se ejecuta siempre que no ocurran excepciones
- catch
  - puede capturar varias excepciones
    - con varios bloques **catch**
- finally
  - se ejecuta siempre antes de salir de try/catch/finally
    - se suele utilizar para hacer limpieza y cerrar recursos
    - se ejecuta siempre, aunque la salida sea por
      - excepcion, break, continue, return

Ejemplo: try/catch/finally

- PrintArgs5: imprime todos los argumentos
  - A traves de varios caminos de ejecución

```
public class PrintArgs5 {
  public static void main (String arg[]){
    for (int i = 0; i <arg.length; i++) {
      try {
        if (arg[i].equals("hola")) { throw new Exception1();}
        if (arg[i].equals("buenas")){throw new Exception2();}
        System.out.println ("argumento " + i + ": " + arg[i]);
      } catch (Exception1 e1){  System.out.println ("hola");
      } catch (Exception2 e2) { System.out.println ("buenas");
      } finally { System.out.println("esto siempre se ejecuta");}
    }
  }
}
```


Clases abstractas e Interfaces

Elementos abstractos

- Los elementos abstractos de Java contiene partes indefinidas, son
  - Clases abstractas
  - Interfaces
- Permiten definir objetos de forma parcial
  - definen solo las signaturas o interfaces
    - la semántica (código) debe ser definida en clases derivadas
- Un elemento abstracto es también
  - un requerimiento para las clases derivadas

Clase abstracta

- Clase abstracta
  - Clase que posee métodos abstractos
  - Una clase abstracta no se puede instanciar
  - Una subclase de una clase abstracta se puede instanciar
    - si implementa (define) todos los elementos abstractos
- Método abstracto
  - solo se define la cabecera
    - No se define el cuerpo

Ejemplo: clase abstracta

- Una clase derivada de websiteA será instanciable
  - si define un método **print()** completo (con código)

```
public abstract class websiteA {

  String name;                                  // campos
  String url;
  String descr;

  public websiteA(String n, String u, String d){  // constructor
      name = n; url = u; descr = d;
  }

  public abstract void print();          // a definir en subclases
}
```


Interfaz

- Un interfaz es
  - Una pseudoclase completamente abstracta
    - Puede utilizarse como cualquier otro objeto en las clases que usen el interfaz
  - Todos sus elementos deben ser abstractos
- Solo puede contener
  - Constantes
    - Variables de tipo “static final”
  - Métodos abstractos
    - No deben llevar el modificador “abstract”

Ejemplo: Interfaz Shape

- Un interfaz puede considerarse un contrato
  - entre una clase que lo usa y una que lo implementa
- Shape define los métodos que un objeto debe tener para poder calcular su área y perímetro
  - Permite definir procedimientos genéricos
    - Implementar Shape obliga a definir todos los métodos del interfaz

```
public interface Shape {
   public double area();
   public double perimeter();
}
```

Uso de Shape

- Los metodos de shapeProc pueden invocarse sobre cualquier objeto que implemente Shape
  - Es un diseño generico
- shapeProc solo debe tener visibilidad de Shape
  - puede procesar cualquier clase que implemente Shape

```
public class ShapeProc {

   public static double totalArea(Shape s[]) {
      double total = 0;
      for (int i=0; i < s.length; i++)  total +=s[i].area();
      return total;
   }

   public static double totalPerimeter(Shape s[]) {
      double total = 0;
      for (int i=0; i < s.length; i++)  total +=s[i].perimeter();
      return total;
   }
}
```

Implementacion de Shape

```
class Circle implements Shape {
   protected double r;
   protected static final double pi = 3.1416;
   public Circle(double r) { this.r = r; }
   public double area() {return pi*r*r; }
   public double perimeter() { return 2*pi*r;}
   public double radius() { return r;}
}

class Rectangle implements Shape {
   protected double w, h;
   public Rectangle(double w, double h) { this.w=w; this.h=h;}
   public double area() {return w*h; }
   public double perimeter() { return 2*(w+h);}
   public double getWidth() { return w;}
   public double getHeight() { return h;}
}
```


Prueba de ShapeProc

```
public class Shapes {

  public static void main(String argv[]){
     Shape[] shapes = new shape[3];
     shapes[0] = new Circle(2.0);
     shapes[1] = new Circle(3.0);
     shapes[2] = new Rectangle(2.0, 3.0);

     system.out.println(shapeProc.totalArea(shapes));
     system.out.println(shapeProc.totalPerimeter(shapes));

  }
```

Usos de un Interfaz

- Sucedáneo de herencia múltiple
  - Una clase puede implementar uno (o varios) interfaces
    - Además de extender otra clase
  - Un interfaz puede extender otros interfaces
- Contrato entre equipos/objetos
  - Muy útil en ingeniería de software
    - Un grupo/objeto usa el interfaz, otro lo implementa
- Sucedáneo parámetro tipo puntero (tipado)
  - en ingles “Callback”

Ejercicio Java 6

- Definir subinterfaz de “Shape”, de nombre “Body”
  - debe enriquecer Shape con un nuevo método “Volume”
- Definir subclase de “ShapeProc”
  - de nombre “BodyProc” (capaz de calcular volúmenes)
  - “BodyProc” añade un método “totalvolume”
    - que calcula sumas de volúmenes de objetos de tipo “Body”
      - Los objetos de tipo “Body” se pasan como elementos del array que recibe “totalvolume” como parámetro
- Definir clase: “Cylinder” como subclase de “Circle”
  - “Circle” implementa “Body” (añade un campo de altura)
- Definir un programa de prueba similar a “Shapes”

Accesibilidad y Visibilidad

Espacio de nombres

- Package (paquete)
  - Agrupación de: clases, excepciones e interfaces
  - Define un ámbito de visibilidad
- Java es un lenguaje de red
  - Los paquetes están distribuidos por toda la Internet
  - Un programa Java puede utilizar clases residentes en otros ordenadores
    - Carga de Applets, RMI, ...
- Los **nombres** deben ser **únicos en toda la red**
  - En ámbitos locales (ficheros) pueden abreviarse

Creación de nombres unicos

- Direcciones de dominio (invertidas) como prefijo (recomendada por los diseñadores de Java)
  - es.upm.dit.jquemada.scom.website.print()
- Nombre de compañía como prefijo
  - netscape.applets.chess.main()
- Los diseñadores reservaron dos prefijos:
  - **java** para JFC
    - Ejemplo: **java**.system.out.println(...)
  - **sun** para clases propias.
    - Ejemplo: **sun**.nfs....

La sentencia package

- Indica pertenencia de la clase al paquete
  - Debe estar en la primera línea del fichero

```
package netscape.applets.chess

public class chess {
  .......
  element1 e;
  .......
}

class element1 {
  .......
}

class element2 {
  .......
}
```

La sentencia import

- Importa al ámbito de visibilidad del fichero
  - El elemento importado puede referenciarse con nombre corto

```
package netscape.applets.chess

import java.util.hashtable;   // importa clase hashtable

import java.lang.*;
              // importa las definiciones publica de java.lang
              // normalmente clases, interfaces y/o excepciones

public class chess {
  .......
  hashtable h = new hashtable();
  .....
}
```


Accesibilidad y Visibilidad

(Transparencias complementarias)

Estructura de ficheros

- Variable de entorno CLASSPATH
  - indica al interprete donde buscar ficheros ejecutables
  - Definición (Unix)
    - setenv CLASSPATH .:/user/classes:/local/java/classes.zip
  - Definición (Windows):
    - set CLASSPATH .;C:\user\classes;D:\local\java\classes.zip
- Un paquete se almacena en un directorio
  - El nombre se usa como path (relativo a CLASSPATH)
    - Por ejemplo, la clase: netscape.applets.chess
    - se almacena como: netscape/applets/chess.class

Paquete

- Sentencia package
  - indica pertenecia a un paquete
  - debe ser la primera sentencia del fichero
    - salvo comentarios y espacio en blanco
  - El fichero puede definir varias clases (solo una publica)
  - Las clases compiladas deben estar en el directorio del paquete para poder ser ejecutadas
- Ficheros que no incluyen la sentencia package
  - pertenecen a un paquete sin nombre
  - son utiles durante el desarrollo o pruebas
  - los ficheros pueden ejecutarse desde el directorio de trabajo

Importación de nombres

- Nombres completos: pueden ser utilizados sin necesidad de importar
  - Ejemplo: java.applet.Applet
- Sentencia import
  - Debe aparecer justo detrás de la sentencia package
    - salvo comentarios y espacio en blanco
  - Hace visibles los objetos importados sin necesidad de indicar a que paquete pertenecen
    - Ejemplo: Applet
  - Colisiones de nombres importados
    - debe utilizarse el nombre completo

Accesibilidad I

- Un paquete es accesible
  - si sus ficheros y directorios son accesibles
    - permiso de lectura, permiso de acceso a través de red, ...
- Clases e interfaces de un paquete
  - son accesibles a todos los paquetes e interfaces del mismo paquete
- Clase e interfaz públicos
  - accesibles en otros paquetes
- Clase e interfaz no-públicos
  - No accesible en otros paquetes

Accesibilidad II

- Miembros de una clase no-privados
  - accesibles desde otras clases del paquete
- Miembros privados de una clase
  - accesibles solo desde la misma clase
- Miembros de la clase A
  - accesibles desde la clase B de otro paquete si
    - A es publico y el miembro es publico
    - A es publico, miembro es protegido (protected) y B es una subclase de A
- Todos los miembros de una clase son accesibles desde dentro de la clase

Resumen modificadores

(Transparencias complementarias)

Modificadores

- Un "modificador" es un atributo de una definición de
  - Clase
  - interfaz
  - campo
  - metodo
  - inicializador
- Define características asociadas a dicha definición
  - La casuística de uso es compleja

Modificadores I

- static: atributo de clase
  - clase (desde Java 1.1): clase de ámbito máximo
  - campo: campo de clase, el mismo en todos los objetos
  - método: método de clase, solo accede a campos static
  - inicializador: se ejecuta al cargar la clase
- final: no modificables (por herencia)
  - clase: no puede tener subclases
  - campo, variable(desde Java 1.1): valor no modificable
  - método: no se puede tapar
    - se garantiza que nadie va a modificar la semantica

Modificadores II

- abstract: con métodos no implementados
  - clase: contiene métodos no implementados
  - interfaz: es abstracto por definición, es opcional
  - método: el método no tiene cuerpo, solo signatura
    - debe pertenecer a una clase abstracta
- synchronized (método): garantiza exclusión mutua
- native (metodo): implementado en C, C++, ASM, ...
- transient (campo): no serializar
- volatile (campo): no optimizar, acceso no sincr.

Modificadores III (metodos y campos)

- Public: visibles en cualquier parte
- Private: Visibles solo dentro de la clase
- Protected: Visibles en paquete y subclases
- "Sin atributo": Visibles en paquete

Java API - JFC

JFC - Java Foundation Classes

- JFC: conjunto de paquetes normalizados
  - Conocidos como: Core Java API o JFC
  - dan un interfaz normalizado de acceso a multiples servicios
- JFC esta en evolución
  - en Java 1.0 habia 7 paquetes y 211 clases
  - en Java 1.1 hay 23 y 503 clases
  - en Java 1.2 aumenta el numero
- La evolución de Java afecta fundamentalmente a JFC
  - Microsoft a creado MFC compatibles con ActiveX

JFC 1.1

- **java.applet**
- + java.awt.datatransfer
- + java.awt.event
- **java.awt**
- java.awt.image
- + java.beans
- **java.io**
- **java.lang**
- + java.lang.reflect
- + java.math
- **java.net**
- + **java.rmi.dgc**
- + **java.rmi**
- + **java.rmi.registry**
- + **java.rmi.server**
- + java.security.acl
- + java.security
- + java.security.interfaces
- + java.sql
- + java.text
- java.util
- + java.util.zip

+ indica que el paquete es nuevo en 1.1 o que tiene nuevos metodos

Paquete: java.lang

Interfaces
- **Cloneable**
- **Runnable**

Clases
- Boolean
- Byte
- Character
- **Class**
- **ClassLoader**
- **Compiler**
- Double
- Float
- Integer
- Long
- Math
- Number
- **Object**
- **Process**
- **Runtime**
- **SecurityManag.**
- Short
- String
- StringBuffer
- **System**
- **Thread**
- **ThreadGroup**
- **Throwable**
- **Void**

Excepciones
- ArithmeticException
- ArrayIndexOutOfBoun.
- ArrayStoreException
- ClassCastException
- ClassNotFoundExcep.
- CloneNotSupportedEx.
- **Exception**
- IllegalAccessEx.
- IllegalArgumentExcep.
- IllegalMonitorStateEx.
- IllegalStateException
- IllegalThreadStateEx.
- IndexOutOfBoundsEx.
- InstantiationException
- InterruptedException
- NegativeArraySizeEx.
- NoSuchFieldException
- NoSuchMethodExcep.
- NullPointerException
- NumberFormatExcep.
- RuntimeException
- SecurityException
- StringIndexOutOfBoun.

Errores
- AbstractMethodError
- ClassCircularityError
- ClassFormatError
- **Error**
- ExceptionInInitializerEr.
- IllegalAccessError
- IncompatibleClassCha.
- InstantiationError
- InternalError
- LinkageError
- NoClassDefFoundError
- NoSuchFieldError
- NoSuchMethodError
- OutOfMemoryError
- StackOverflowError
- ThreadDeath
- UnknownError
- UnsatisfiedLinkError
- VerifyError
- VirtualMachineError

Clase java.lang.System

- Clase que modela el sistema sobre el que esta ejecutandose el programa Java
- No puede instanciarse
  - Solo puede utilizarse
- Solo existe una instancia que es creada por la maquina virtual
- Da acceso a los recursos del sistema
  - Entrada/salida
  - Librerias, Ficheros
  - .....

java.lang.System I

```
public final class System extends Object {
   // No Constructor
   // Constants
        public static final PrintStream err;
        public static final InputStream in;
        public static final PrintStream out;
   // Class Methods
        public static native void arraycopy(Object src,
              int src_position, Object dst, int dst_position, int length);
        public static native long currentTimeMillis();
        public static void exit(int status);
        public static void gc();
        public static Properties getProperties();
        public static String getProperty(String key);
        public static String getProperty(String key, String def);
        .........
```

java.lang.System II

```
        .............
        public static SecurityManager getSecurityManager();
        public static String getenv(String name);
        public static native int identityHashCode(Object x);
        public static void load(String filename);
        public static void loadLibrary(String libname);
        public static void runFinalization();
        public static void runFinalizersOnExit(boolean value);
        public static void setErr(PrintStream err);
        public static void setIn(InputStream in);
        public static void setOut(PrintStream out);
        public static void setProperties(Properties props);
        public static void setSecurityManager(SecurityManager s);
}
```

Java API - JFC

(Transparencias complementarias)

java.lang.String I

```
public final class String extends Object implements Serializable {
    // Public Constructors
        public String();
        public String(String value);
        public String(char[] value);
        public String(char[] value, int offset, int count);
        public String(byte[] ascii, int hibyte, int offset, int count);
        public String(byte[] ascii, int hibyte);
        public String(byte[] bytes, int offset, int length, String enc) throws UnsupportedEncodingEx.;
        public String(byte[] bytes, String enc) throws UnsupportedEncodingException;
        public String(byte[] bytes, int offset, int length);
        public String(byte[] bytes);
        public String(StringBuffer buffer);
    // Class Methods
        public static String copyValueOf(char[] data, int offset, int count);
        public static String copyValueOf(char[] data);
        public static String valueOf(Object obj);
        public static String valueOf(char[] data);
        public static String valueOf(char[] data, int offset, int count);
        public static String valueOf(boolean b);
        public static String valueOf(char c);
        public static String valueOf(int i);
        public static String valueOf(long l);
        public static String valueOf(float f);
        public static String valueOf(double d);
    // Public Instance Methods
        public char charAt(int index);
        public int compareTo(String anotherString);
        public String concat(String str);
        public boolean endsWith(String suffix);
        public boolean equals(Object anObject);  // Overrides Object
```


java.lang.String II

```
        public boolean equalsIgnoreCase(String anotherString);
        public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin);
        public byte[] getBytes(String enc) throws UnsupportedEncodingException;
        public byte[] getBytes();
        public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin);
        public int hashCode();  // Overrides Object
        public int indexOf(int ch);
        public int indexOf(int ch, int fromIndex);
        public int indexOf(String str);
        public int indexOf(String str, int fromIndex);
        public native String intern();
        public int lastIndexOf(int ch);
        public int lastIndexOf(int ch, int fromIndex);
        public int lastIndexOf(String str);
        public int lastIndexOf(String str, int fromIndex);
        public int length();
        public boolean regionMatches(int toffset, String other, int ooffset, int len);
        public boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len);
        public String replace(char oldChar, char newChar);
        public boolean startsWith(String prefix, int toffset);
        public boolean startsWith(String prefix);
        public String substring(int beginIndex);
        public String substring(int beginIndex, int endIndex);
        public char[] toCharArray();
        public String toLowerCase(Locale locale);
        public String toLowerCase();
        public String toString();  // Overrides Object
        public String toUpperCase(Locale locale);
        public String toUpperCase();
        public String trim();
}
```

Clase java.lang.Class

- java.lang.Class modela el espacio de tipos
  - Las instancias de java.lang.Class representan los tipos del programa Java en ejecución
    - Incluye: clases, arrays, tipos primitivos y la palabra reservada void
- No tiene constructores publicos utilizables
  - Las instancias de esta clase las crea la maquina virtual Java en el “cargador de clases” llamando a “defineClass(..)”

java.lang.Class I

```
public final class Class extends Object implements Serializable {
  // No Constructor
  // Class Methods
    public static native Class forName(String className)
                                        throws ClassNotFoundException;
  // Public Instance Methods
    public native ClassLoader getClassLoader();
    public Class[] getClasses();
    public native Class getComponentType();
    public Constructor getConstructor(Class[] parameterTypes)
                    throws NoSuchMethodException, SecurityException;
    public Constructor[] getConstructors() throws SecurityEx.;
    public Class[] getDeclaredClasses() throws SecurityExcep.;
    public Constructor getDeclaredConstructor(Class[]
            parameterTypes) throws NoSuchMethodEx., SecurityEx.;
    .......
```

java.lang.Class II

```
    .........
    public Constructor[] getDeclaredConstructors()
                                              throws SecurityException;
    public Field getDeclaredField(String name) throws
                            NoSuchFieldException, SecurityException;
    public Field[] getDeclaredFields() throws SecurityException;
    public Method getDeclaredMethod(String name, Class[]
      parameterTypes) throws NoSuchMethodExcep., SecurityExcep.;
    public Method[] getDeclaredMethods() throws SecurityExcep.;
    public Class getDeclaringClass();
    public Field getField(String name)
                    throws NoSuchFieldException, SecurityException;
    public Field[] getFields() throws SecurityException;
    public native Class[] getInterfaces();
    public Method getMethod(String name, Class[] parameterTypes)
                  throws NoSuchMethodException, SecurityException;
    .........
```


java.lang.Class III

```
    .........
    public Method[] getMethods() throws SecurityException;
    public native int getModifiers();
    public native String getName();
    public URL getResource(String name);
    public InputStream getResourceAsStream(String name);
    public native Object[] getSigners();
    public native Class getSuperclass();
    public native boolean isArray();
    public native boolean isAssignableFrom(Class cls);
    public native boolean isInstance(Object obj);
    public native boolean isInterface();
    public native boolean isPrimitive();
    public native Object newInstance()
                throws InstantiationException, IllegalAccessException;
    public String toString();  // Overrides Object
}
```

Canales de Comunicación

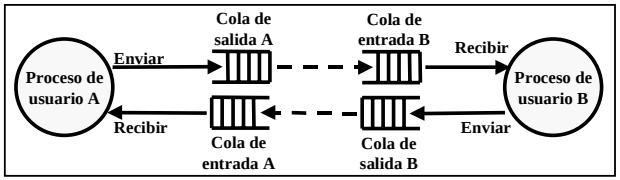
Bibliografía:

- Java Network Programing, Hughes et all, Cap. 4
- Documentacion API: paquete java.io

Canales de comunicación

- Estructuras de datos a través de las cuales se puede intercambiar información
  - Posee memoria para almacenar los mensajes hasta su recepción por el usuario
    - Dos operaciones : enviar y recibir
- Tratamiento uniforme de la comunicación local y remota
  - Muy utilizado en aplicaciones distribuidas
- Existen múltiples variantes
  - Unidireccionales, multipunto, ...

Ejemplo de canal con memoria



- Operaciones de acceso
  - Enviar: escribir en cola
  - Recibir: leer de cola

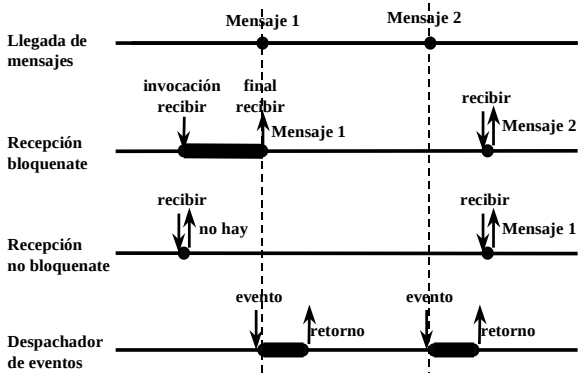
Estados de la cola:

- **Llena:**
  - **no se puede enviar**
- **Vacía:**
  - **no se puede recibir**
- **Semillena:**
  - **se puede enviar y rec.**

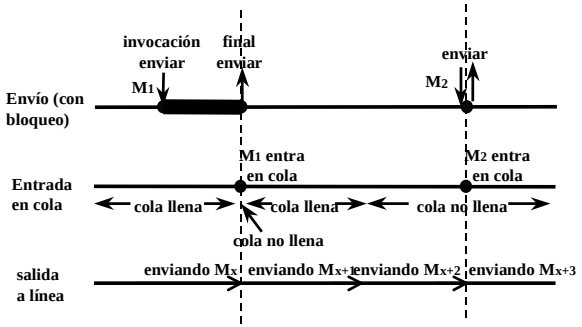
Acceso a un canal

- Acceso bloqueante
  - **enviar**: bloquea el proceso hasta el envío del paquete
  - **recibir**: bloquea el proceso hasta recibir un paquete
- Acceso no bloqueante
  - Las operaciones **enviar** o **recibir** no esperan
  - Ejemplo: si **recibir** no encuentra un paquete devuelve un código de retorno, sin esperar a que llegue un paquete
- Despachador de eventos (event dispatching)
  - Invoca **procedimientos de atención a los eventos**
  - Eventos: recepción de paquete, vence temporizador, ...
  - Similar a un mecanismo de interrupciones software

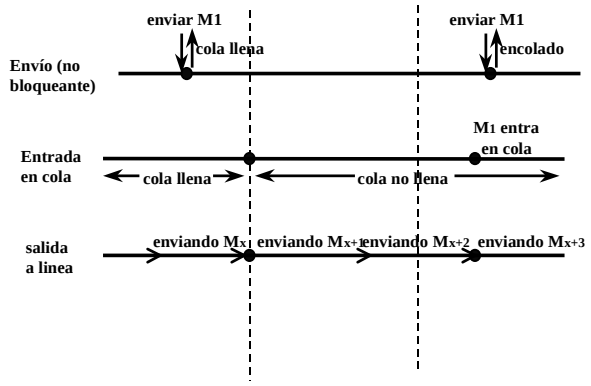
Ejemplo: recepción de mensajes



Ejemplo: envio con bloqueo



Ejemplo: envío no bloqueante



Flujos (Streams)

Bibliografía:

- Java Network Programing, Hughes et all, Cap. 4
- Documentacion API: paquete java.io

Streams en Java

- Los canales de comunicación en Java se modelan como objetos derivados de la clase **Stream**
  - Stream es una clase abstracta
    - Define: estructura genérica de acceso secuencial a información
      - La información se estructura como secuencias de bytes
- Acceso a dispositivos se modela
  - con clases derivadas de Stream
  - Objeto Stream: descriptor de acceso a canal
    - Manejo: creación, uso y destrucción
- Están definidos en el paquete: **java.io**

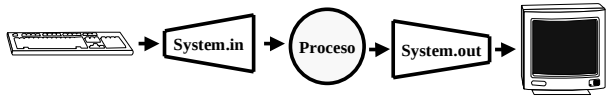
Canales

- Un canal de comunicación se gestiona con dos descriptores:
  - de lectura (InputStream)
  - de escritura (OutputStream)
    - Un canal bidireccional consta de dos canales unidirecionales

- Acceso (normalmente):
  - bloqueante: se espera a la finalización de la operación
  - secuencial: la información se lee en el orden introducido
  - La unidad de información es el byte

Ejemplos de flujos de datos

- Flujos: modelan el acceso a recursos
  - E/S, manejo de ficheros, comunicación entre hebras o procesos, ...
- Flujos del sistema
  - System.in: entrada estandar (de teclado)
  - System.out: salida estandar (a pantalla)
  - System.err: salida de error (a pantalla)
    - son variables de tipo stream de la clase “java.lang.system”



Ejemplo: Entrada/salida

```
import java.io.*;

class charIO {
   public static void main(String args[])  throws IOException {
      int charRead;
      while ((charRead = System.in.read ()) >= 0)
          System.out.write(charRead);
      System.out.println("FIN");
   }
}
```

InputStream (I)

- InputStream: clase abstracta
  - define los métodos minimos de cualquier InputStream
- La implementación se basa en “read()”
  - read() es abstracta (no esta implementada) y se implementa en las subclases

```
public abstract class InputStream extends Object  {
   .......
   public abstract int read() throws IOException  ;  lee un byte
   public int read(byte b[]) throws IOEx. ; lee hasta  “b.length”
   public int read(byte b[], int off, int len) throws IOException
                  ; lee hasta “len” bytes con desplazamiento “off”
   public void close() throws IOEx.  ; cierra y libera recursos
   .....
```

Métodos de "InputStream" (II)

- Funciones especiales: gestión de la cola de lectura
  - Eliminación de partes, análisis de estado, introducción de marcas y vuelta a atrás.

```
  .......
  public long skip(long n) throws IOExcep. ; elimina n bytes
  public int available() throws IOEx.  ; num. bytes disponibles
  public synchronized void mark(int readlimit) ;marca de sincr
  public synchr. void reset() throws IOExcep.  ;vuelve a  marca
  public boolean markSupported()     ; indica si soporta marca
}
```

Características* de Read() y Write()

- Read()
  - devuelve el byte leído en un entero
    - devuelve un -1 si se alcanza el final de un stream
  - bloquea hasta:
    - 1) tener bytes de entrada disponibles
    - 2) final de stream
    - 3) activación de una excepción
- Write()
  - Aunque es poco frecuente, se puede bloquear si el buffer esta lleno

*Puede haber excepciones

Jerarquia de InputStream

- Deriva los diversos tipos de streams
  - Cada clase maneja un tipo de flujos o de dispositivos
- Existe otra similar de OutputStreams

```
InputStream  ; Superclase de la jerarquía (abstracta)

  ByteArrayInputStream ; Arrays de bytes (formateo)
  FilterInputStream    ; Superclase: filtros de streams
    ......             ;
                       ;
  FileInputStream      ; Lectura de ficheros
  PipedInputStream     ; Comunica hebras con excl. mutua
  SequenceInputStream  ; Combina varios  streams
  ObjectInputStream    ; Deserializa objetos
  StringBufferInputStream ; deprecada en Java 1.1
  .......
```

Métodos de "OutputStream"

- OutputStream: clase abstracta
  - define los métodos minimos de cualquier OutputStream
- La implementación se basa en "write()"
  - write() es abstracta (no esta implementada) y se implementa en las subclases

```
public abstract class OutputStream extends Object  {
   ......
   public abstract void write(int b) throws IOExcep. ; 1 byte
   public void write(byte b[]) throws IOEx.  ; "b.length" bytes
   public void write(byte b[], int off, int len) throws IOExcep.
                          ; escribe hasta "len" bytes con despl. "off"
   public void flush() throws IOException     ; fuerza el envio
   public void close() throws IOException     ; libera recursos
}
```

Gestión y Manejo de Ficheros

Bibliografía:

- Java Netw. Programing, Hughes et all, Capítulo 5
- Documentación API: paquete java.io

Manejo de Ficheros

- Los ficheros se crean y manipulan a través de objetos de acceso
  - Creación del objeto
    - Creación o apertura del fichero
  - Operaciones read() y write() sobre el objeto
    - lectura y escritura en el fichero
  - Operación close()
    - Cierre del fichero
- java.io incluye
  - ficheros de acceso **secuencial**
  - ficheros de acceso **aleatorio**

Acceso secuencial y aleatorio

- Fichero de acceso **secuencial**
  - Clase FileOutputStream (escritura)
    - Creación del objeto:
      - El fichero se abre en modo escritura al crear el objeto
        - » si no existe **se crea**
  - Clase FileInputStream (lectura)
    - Creación del objeto:
      - El fichero se abre en modo lectura al crear el objeto
        - » si no existe **se activa una excepción**
- Fichero de acceso **aleatorio**
  - clase RandomAccessFile
    - Permite acceso en modo lectura y/o escritura
    - Permite acceso a cualquier byte, sin haber leído los anteriores

Ejemplo: copy

- Programa de copia de ficheros
  - Invocación: copy <src> <dest>
  - debe copiar el contenido del fichero <src> en <dest>
    - si la sintaxis de invocación no es correcta debe indicarlo
    - si <dest> no existe debe crearlo
    - si <src> no existe debe generar una excepción

Ejemplo: Copy <src> <dest>

```
import java.io.*;
class copy {
   public static void main (String args[]) throws Exception {
      if (args.length != 2)
         throw (new Exception ("Syntax: Copy <src> <dest>"));
      FileInputStream in = new FileInputStream(args[0]);
      FileOutputStream out = new FileOutputStream(args[1]);

      byte buffer[] = new byte[16];
      int n;
      while ((n = in.read (buffer)) > -1)
         out.write(buffer, 0, n);
      out.close();   in.close();
  }
}
```

Ejemplo: FileInputStream

```
public class FileInputStream extends InputStream {
 ......
 public FileInputStream(String n)throws FileNotFoundEx.
 public FileInputStream(File file) throws FileNotFoundEx.
 public FileInputStream(FileDescriptor fdObj)
 .......
 public native int read() throws IOException
 public int read(byte b[]) throws IOException
 public int read(byte b[], int off, int len) throws IOException
 public native long skip(long n) throws IOException
 public native int available() throws IOException
 public native void close() throws IOException
 public final FileDescriptor getFD() throws IOException
 protected void finalize() throws IOException
}
```


Ejemplo: FileOutputStream

```
public class FileOutputStream extends OutputStream {
 ........
 public FileOutputStream(String name) throws IOException
 public FileOutputStream(String name,
                         boolean append) throws IOException
 public FileOutputStream(File file) throws IOException
 public FileOutputStream(FileDescriptor fdObj)
 .........
 public native void write(int b) throws IOException
 public void write(byte b[]) throws IOException
 public void write(byte b[], int off, int len) throws IOExcep.
 public native void close() throws IOException
 public final FileDescriptor getFD() throws IOException
 protected void finalize() throws IOException
}
```

Ejemplo: append

- Transformar el programa “copy <src> <dest>”
  - en un programa “append <src> <dest>” que añada “<src>” a “<dest>” a partir del ultimo byte de “<dest>”
- Las clases File In(Out)putStream solo pueden manejar ficheros de forma sequencial.
  - Para acceder a ficheros de forma aleatoria se necesita la clase **RandomAccessFile**
- Se recomienda
  - abrir “<dest>” como RandomAccessFile
  - posicionar el puntero de acceso al final
  - escribir el contenido nuevo (usar código de copy)

Ejemplo: append <src> <dest>

```
import java.io.*;
class append {
   public static void main (String args[]) throws Exception {
      if (args.length != 2)
          throw(new Exception("Syntax: Append <src><dest>"));
      FileInputStream in = new FileInputStream(args[0]);
      RandomAccessFile out =
                          new RandomAccessFile(args[1],"rw");
      out.seek(out.length());

      byte buffer[] = new byte[16];
      int n;
      while ((n = in.read (buffer)) > -1)
         out.write(buffer, 0, n);
      out.close();   in.close();
  }
}
```

Ejercicio: Java 7

- Realizar un programa “merge”, que se debe invocar con la siguiente sintaxis
  - java “merge <dest> <f1> <f2> .. <fn>”
- El programa debe crear un fichero <dest> con la concatenación de los contenidos de <f1> a <fn>, siendo n un numero variable

Filtros de Streams

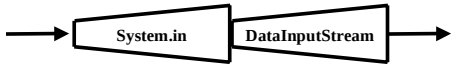
Jerarquía de FilterStream (entr.)

Los “FilterStreams” permiten encadenar filtros en serie que procesan la información de un flujo.

FilterInputStream	; superclase de filtros
BufferedInputStream	; con buffer intermedio ; para mayor eficiencia
CheckedInputStream	; integridad (checksum)
DataInputStream	; serializa tipos primitivos ; en secuencias de bytes
InflaterInputStream	; decompresión de flujo
GZIPInputStream	; decompresión GZIP
ZipInputStream	; decompresión Zip
LineNumberInputStream	; no recomendada en 1.1
PushbackInputStream	; lectura, vuelta atrás

Ejemplo:

- DataInputStream permite lectura de texto
  - readline() esta deprecada, pero se utiliza por sencillez




```
import java.io.*;
class lineIO {
  public static void main(String args[])  throws IOException {
    String line;
    DataInputStream lineIn =
                        new DataInputStream(System.in);
    while (!(line = lineIn.readLine ()).equals(“.”))
        System.out.println(line);
     System.out.println(”END");
  }
}
```

Internacionalización: Codigos de caracteres, Readers y Writers

Bibliografía:

- Java Netw. Programing, Hughes et all, Apendice C
- Documentación API: paquete java.io

Internacionalización

- Capacidad de un programa para manejar **interfaces de usuario** utilizables en **múltiples países**
- Características
  - Deben poder representar todas las lenguas
    - con diferentes alfabetos o jeroglíficos
  - Deben poder utilizar formatos locales en
    - Fechas, nombres, números, ...

Internacionalización en Java

- Java soporta internacionalización de programas
- Facilidades
  - Código Unicode
    - permite representar múltiples alfabetos
  - Readers y Writers (clases de java.io)
    - manejadores especiales para flujos de caracteres
  - java.text:
    - paquete de manejo de convenciones locales para las lenguas mas comunes
  - java.util.ResourceBoundle:
    - clase de manejo de bloques de mensajes en múltiples lenguas

Codificación de caracteres

- Codificación internacionalizada
  - Debe soportar las lenguas existentes
    - Tipos de lenguas: alfabéticas y jeroglíficas
- Lenguas alfabéticas:
  - Codificación indirecta basada en un alfabeto
    - Palabra: se representa con un bloque de símbolos del alfabeto
    - Suelen tener un numero de símbolos pequeño (entre 20 y 100)
  - Ejemplo: lenguas de origen latino, griego o árabe
- Lenguas jeroglíficas:
  - Poseen un símbolo diferente para cada palabra
    - Necesitan un conjunto de símbolos muy grande
  - Ejemplo: chino(> 80000 símbolos), japonés o coreano

Códigos mas usados

- Mundo occidental/anglosajón:
  - ASCII, ISO Latin-1, UNICODE, UTF8
- ASCII:
  - codifica ingles-americano
    - Además incluye caracteres de control de teletipos
  - Formato: 7 bits por caracter
- ISO Latin-1:
  - extiende ASCII con 1 bit
  - Codifica caracteres de otras lenguas de raíz latina
    - incluye castellano, catalán, ...

Código Unicode

- UNICODE:
  - Código pseudointernacionalizado
    - Mas información en: http://www.unicode.org
  - extiende ISO Latin-1
  - Formato de longitud fija: 2 bytes
    - Cuando el primer byte es \h00 el otro byte es ISO Latin-1
  - Alfabetos soportados:
    - Europa, Africa y parte de Asia (subconjunto de chino, japones,..)
  - Alfabetos no soportados:
    - Braille, Cherokee, Etíope, Mongol, Javanés, ..
  - Solo ocupa 38.885 combinaciones de 65536
    - Siguen incorporandose mas lenguas

Codigo UTF8

- UTF8: Universal Character Set Transformation Format 8-bit form
  - optimizado para representar caracteres ASCII.
- Formato de longitud variable (1 a 4 bytes)
  - 1 byte: incluye ASCII en últimos 7 bits, excepto null
    - formato: 0xxxxxxx
  - 2 bytes: códigos 127 a 2048 de Unicode (11 bits)
    - formato: 110xxxxx10xxxxxx
  - 3 bytes: resto de UNICODE y otros
    - formato: 1110xxxx10xxxxxx10xxxxxx
  - Otros códigos: 4 bytes
    - Para japonés, chino y coreano por separado
    - formato: 1110xxxx10xxxxxx10xxxxxx10xxxxxx

Representación de caracteres en Java

- Java ha sido diseñado para soportar internacionalición
  - La clase “char” de Java utiliza UNICODE
- Java soporta conversión a (pseudo) UTF8
  - para optimizar almacenamiento de texto en inglés
    - Un texto inglés en UNICODE ocupa el doble que en UTF8
  - UTF8 de Java difiere ligeramente de la norma
    - null (ocupa 2 bytes): 1100000010000000
    - No soporta códigos de 4 bytes

Tipos primitivos: char

```
char first = ‘A’;       // crea first y asigna caracter: A

char c = ‘”’;           // crea c y asigna caracter: ”
c = ‘\u0022’;           // asigna caracter: ”
c = ‘\042’;             // asigna caracter: ”
c = ‘\”’;               // asigna caracter: ”
      // 22 (hexadecimal) = 42 (octal) = código de caracter ”

       //  \b    espacio atrás
       //  \t    tabulador horizontal
       //  \n    linea nueva
       //  \r    retorno de carro
       //  \f    paso de pagina
       //  \”    comillas
       //  \’    comilla
       //  \\    barra hacia atras
```


Problema: múltiples códigos

- Dualidad de códigos en S.O. y Java
  - Casi ningún S.O. utiliza UNICODE
    - Java se diseñó para aceptar programas en ASCII o ISO Latin-1
  - Los editores suelen usar ISO Latin-1 o ASCII
    - Los programas fuente de Java suelen estar en ISO Latin-1
- Recomendaciones sobre portabilidad
  - Los compiladores (normalmente) esperan que el programa esta almacenado en ISO Latin-1/ASCII
    - Todas las palabras reservadas de Java se codifican en ASCII
  - Se recomienda escribir los programas de Java en ASCII
    - Por ejemplo: para portabilidad de PC a Mac sin problemas

Readers y Writers

- Manejadores de flujos de caracteres
  - Soporta adecuadamente la conversión de código
    - entre S.O. y Java (Unicode)
    - Aparecen en Java 1.1
  - **Readers**: lectores del exterior
    - traducen de código de S.O. a Unicode
  - **Writers**: escriben en el exterior
    - traducen de Unicode a código de S.O.
- Streams:
  - poseen algunos métodos (deprecados) para traducción de ASCII/ISO Latin-1 a Unicode
    - No se recomienda su utilización

Conversión entre Reader/Writer y Stream

- Un Stream y un sistema operativo maneja habitualmente bytes como la unidad básica.
- Los caracteres se codifican como bytes
  - **System.in** y **System.out** son de tipo **Stream**
- La conversión a UNICODE hace necesario utilizar conversores de **Stream** a **Reader/Writer**

```
BufferedWriter out =
    new BufferedWriter(new OutputStreamWriter(System.out));
BufferedReader in =
    new BufferedReader(new InputStreamReader(System.in));
```

Ejemplo: Entrada/salida


```
import java.io.*;

public class reader {
  public static void main (String args[]) throws IOException {
    String line;
    LineNumberReader r;
    r = new LineNumberReader(
                              new InputStreamReader(System.in));
    line = r.readLine();
    System.out.println("Did you input? " + line);
  }
}
```


Jerarquía de Readers (input)

- La jerarquía de “Readers” aparece en Java 1.1
  - Soporta flujos de caracteres internacionalizados
  - Basada en UNICODE.
- Existe otra jerarquía equivalente de “Writers”

Reader ; para leer flujos de caracteres
CharArrayReader ; arrays de caracteres
FilterArrayReader ; superclase de filtros
PushbackReader ; con vuelta atrás
BufferedReader ; con buffer intermedio
LineNumberReader ; entrada de texto
InputStreamReader ; conversión de bytestream
FileReader ; lectura de ficheros de carac.
PipedReader ; pipes de caracteres
StringReader ; conversión de strings

Ejercicio: Java 8

- Realizar un programa transcodificador de ficheros
  - Invocación: java “ascii2unicode <dest> <source>”
  - Formato de entrada: secuencia de bytes
    - Cada byte es un caracter en ASCII/ISO Latin 1
      - tal y como almacena el S.O.
  - Formato de salida: secuencia de bytes
    - cada caracter ASCII/ISO Latin 1 de entrada, se sustituye por dos bytes equivalentes del código Unicode
  - Utilizar “**LineNumberReader” para leer texto**
- Realizar también la transcodificación inversa